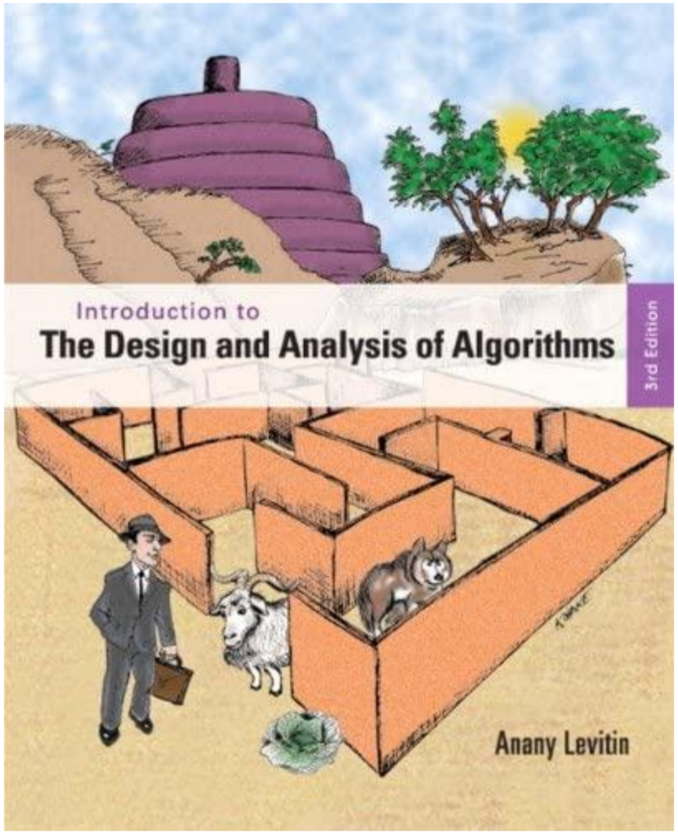
Design and Analysis of Computer Algorithm

Acknowledgement

- This lecture note has been summarized from lecture note on Data Structure and Algorithm, Design and Analysis of Computer Algorithm all over the world. I can't remember where those slide come from. However, I'd like to thank all professors who create such a good work on those lecture notes. Without those lectures, this slide can't be finished.

More Information

- Textbook
  - *Anany Levitin,* ***Introduction to the Design and Analysis of Algorithms,*** *3/e, Pearson, 2012*.



Course Objectives

- This course introduces students to the analysis and design of computer algorithms. Upon completion of this course, students will be able to do the following:
  - Analyze the asymptotic performance of algorithms.
  - Demonstrate a familiarity with major algorithms and data structures.
  - Apply important algorithmic design paradigms and methods of analysis.
  - Synthesize efficient algorithms in common engineering design situations.

What is Algorithm?

- Algorithm
  - is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
  - is thus a sequence of computational steps that transform the input into the output.
  - is a tool for solving a well - specified computational problem.
  - Any special method of solving a certain kind of problem (Webster Dictionary)

What is a program?

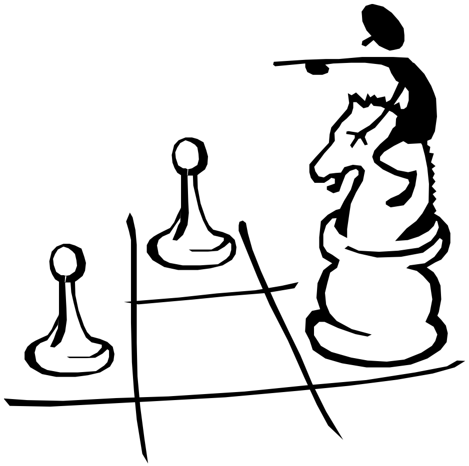
- A program is the expression of an algorithm in a programming language
- a set of instructions which the computer will follow to solve a problem

Where We're Going (1/2)

- Learn general approaches to algorithm design
  - Divide and conquer
  - Greedy method
  - Dynamic Programming
  - Graph Theory
  - Dynamic Programming

Some Application

- Study problems these techniques can be applied to
  - sorting
  - data retrieval
  - network routing
  - Games
  - etc



Importance of Analyze Algorithm

- Need to recognize limitations of various algorithms for solving a problem
- Need to understand relationship between problem size and running time
  - When is a running program not good enough?
- Need to learn how to analyze an algorithm's running time without coding it
- Need to learn techniques for writing more efficient code

What do we analyze about them?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done (complexity)
  - Basic operations to do task
- Amount of space used
  - Memory used

What do we analyze about them?

- Simplicity, clarity
  - Verification and implementation.
- Optimality
  - Is it impossible to do better?

Complexity

- The complexity of an algorithm is simply the amount of work the algorithm performs to complete its task.

Analysis of Algorithms

- Analysis is performed with respect to a computational model
- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

Asymptotic Performance

- In this course, we care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time

Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does $O(n)$ running time mean? $O(n^2)$? $O(n \lg n)$?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to define this notation more formally and completely

Big O Notation

- $O(g(n)) = \{ f(n) \mid \exists\, c > 0$, and n_0 , so that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

- $f(n) = O(g(n))$ means $f(n) \in O(g(n))$ (i.e, at most)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \le c$$

Omega Ω Notation

- $\Omega(g(n)) = \{ f(n) \mid \exists\, c > 0$, and n_0, so that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

- $f(n) = \Omega(g(n))$ means f(n) $\in \Omega(g(n))$ (i.e, at least)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \ge c$$

Theta Θ Notation

- Combine lower and upper bound
  - $f(n)=\Theta(g(n))$ means $f(n)=O(g(n))$ and $f(n) = \Omega(g(n))$

- Means tight: of the same order
- $\Theta(g(n)) = \{ f(n) \mid \exists\ a,b > 0$, and n_0 , so that $0 \le ag(n) \le f(n) \le bg(n)$ for all $n \ge n_0\}$
  - $f(n)=\Theta(g(n))$ means $g(n)=\Theta(f(n))$?

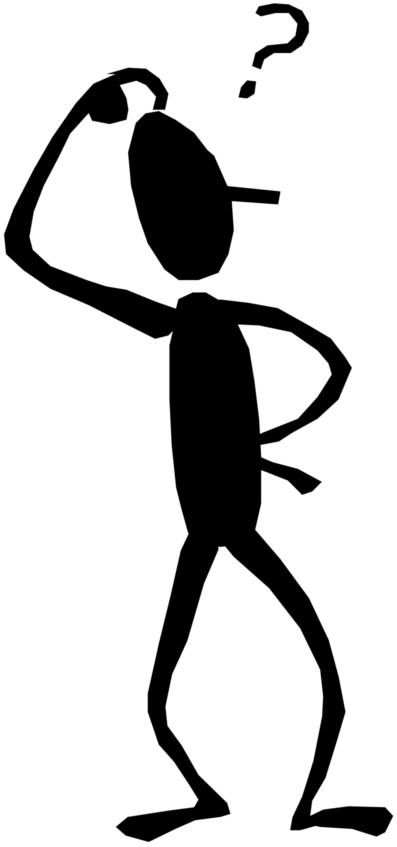
Define Problem

- **Problem**:
  - Description of Input-Output relationship
- **Algorithm**:
  - A sequence of computational step that transform the input into the output.
- **Data Structure:**
  - An organized method of storing and retrieving data.
- **Our task:**
  - Given a problem, design a ***correct*** and ***good*** algorithm that solves it.

Which algorithm is better?

The algorithms are correct, but which is the best?

- Measure the running time (number of operations needed).
- Measure the amount of memory used.
- Note that the running time of the algorithms increase as the size of the input increases.

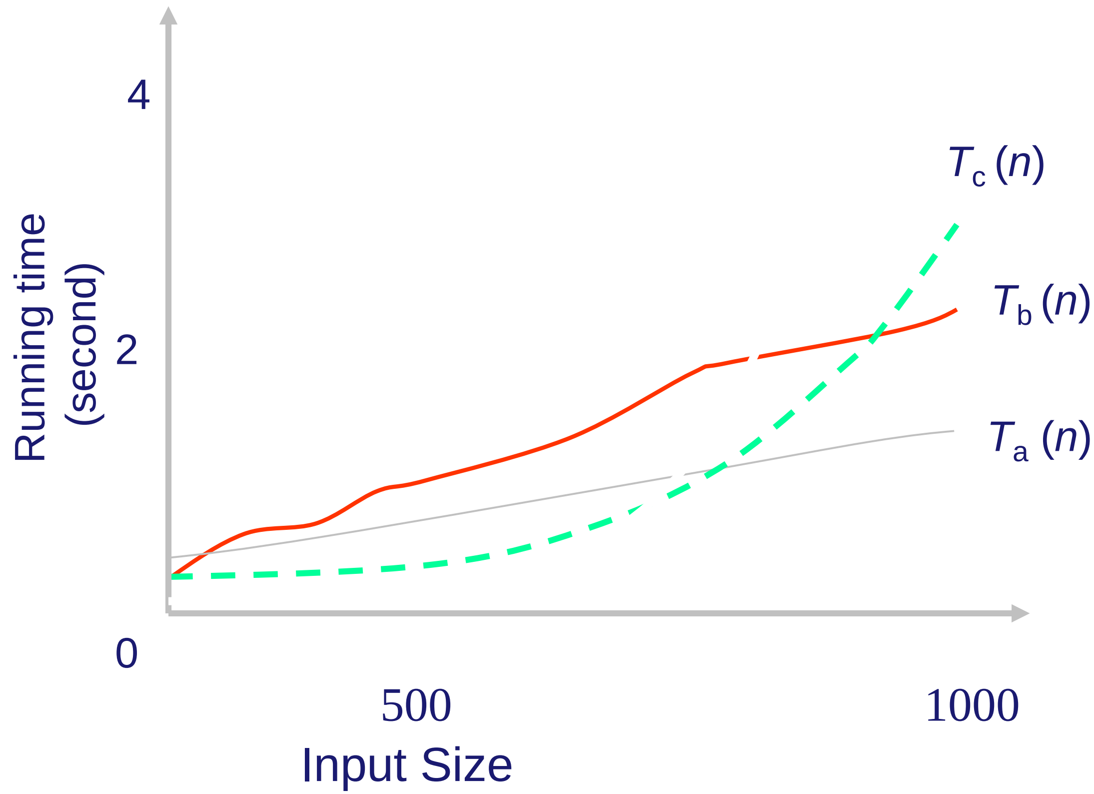


Time vs. Size of Input

Measurement parameterized by the size of the input.

The algorihtms A,B,C are *implemented* and run in a PC.

Let $T_k(n)$ be the amount of time taken by the Algorithm



What is Algorithm Analysis?

- How to estimate the time required for an algorithm

- Techniques that drastically reduce the running time of an algorithm

- A mathemactical framework that more rigorously describes the running time of an algorithm

Review: Asymptotic Performance

- *Asymptotic performance*: How does algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
- Remember that we use the RAM model:
    - All memory equally expensive to access
    - No concurrent operations
    - All reasonable instructions take unit time
        - Except, of course, function calls
    - Constant word size
        - Unless we are explicitly manipulating bits

An Example: Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

$i = \varnothing \quad j = \varnothing \quad key = \varnothing$

$A[j] = \varnothing \quad A[j+1] = \varnothing$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = 2 j = 1 key = 10
A[j] = 30 A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2 j = 1 key = 10
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2 j = 1 key = 10
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$ $j = 0$ $key = 10$
$A[j] = \varnothing$ $A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 0 \quad key = 40$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 0 \quad key = 40$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$ $j = 3$ $key = 20$

$A[j] = 40$ $A[j+1] = 20$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4   j = 3   key = 20
A[j] = 40   A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

$i = 4 \quad j = 2 \quad key = 20$

$A[j] = 30 \quad A[j+1] = 30$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4   j = 1   key = 20

A[j] = 10   A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4   j = 1   key = 20
A[j] = 10   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4   j = 1   key = 20

A[j] = 10   A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4   j = 1   key = 20
A[j] = 10   A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

Done!

Insertion Sort

InsertionSort(*A*, *n*)

for $i = 2$ to n do

$key = A[i]$ —————— c_1

$j = i - 1$

while $j > 0$ and $A[j] > key$ do

$A[j+1] = A[j]$ —————— c_2

$j = j - 1$

$A[j+1] = key$

$$T(n) \leq \sum_{i=2}^{n} (c_1 + c_2 i) = c_3 n^2 + c_4 n + c_5$$

Analysis

- Simplifications
  - Ignore actual and abstract statement costs
  - *Order of growth* is the interesting measure:
    - Highest-order term is what counts
      - Remember, we are doing asymptotic analysis
      - As the input size grows larger it is the high order term that dominates

Upper Bound Notation

- We say InsertionSort's run time is *$O(n^2)$*
  - Properly we should say run time is *in* $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n): \exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$

Insertion Sort Is $O(n^2)$

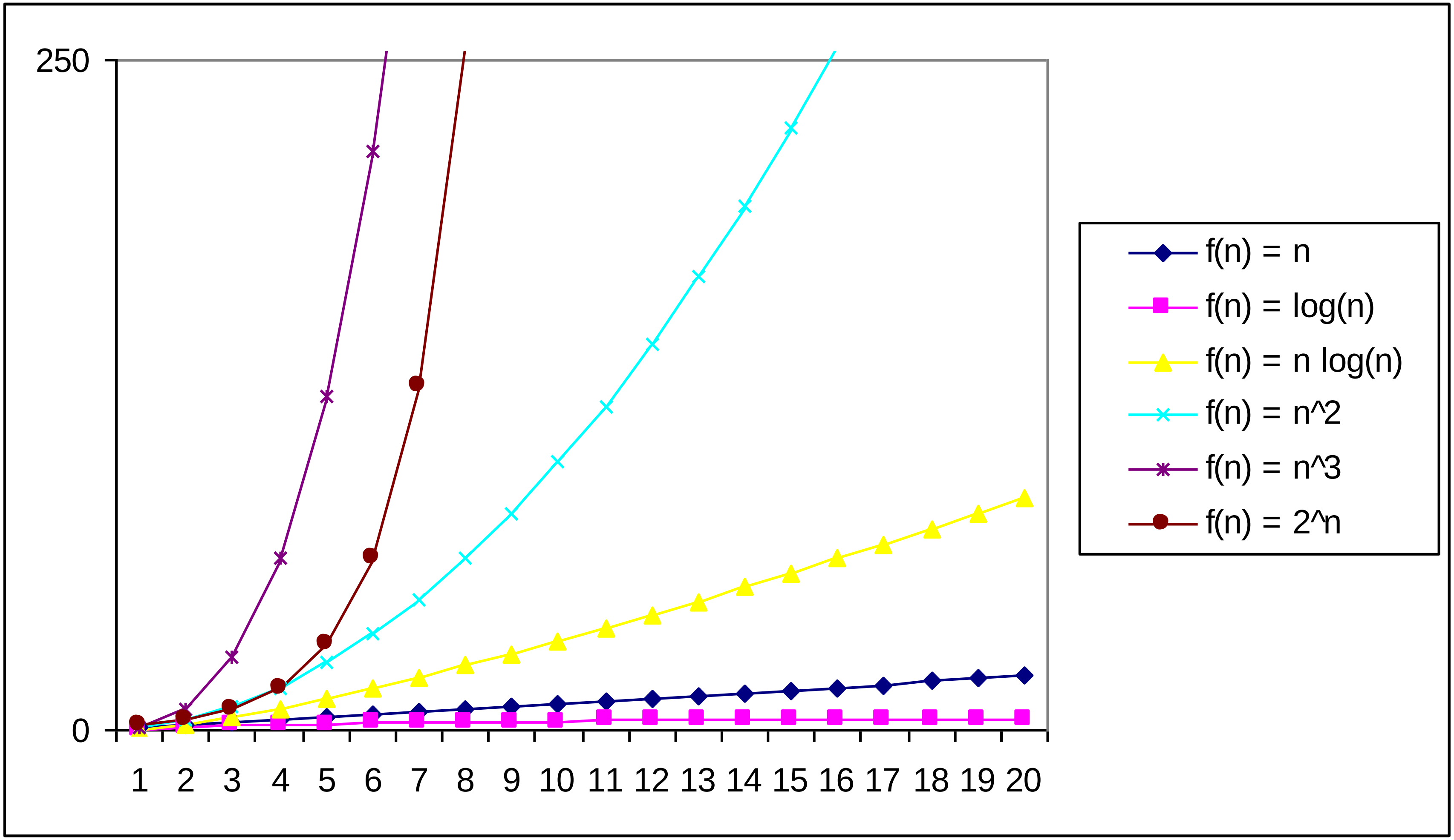
- Proof
  - Suppose runtime is $an^2 + bn + c$
    - If any of a, b, and c are less than 0 replace the constant with its absolute value
  - $an^2 + bn + c \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
  - $\leq 3(a + b + c)n^2$ for $n \geq 1$
  - Let $c' = 3(a + b + c)$ and let $n_0 = 1$

Big O Fact

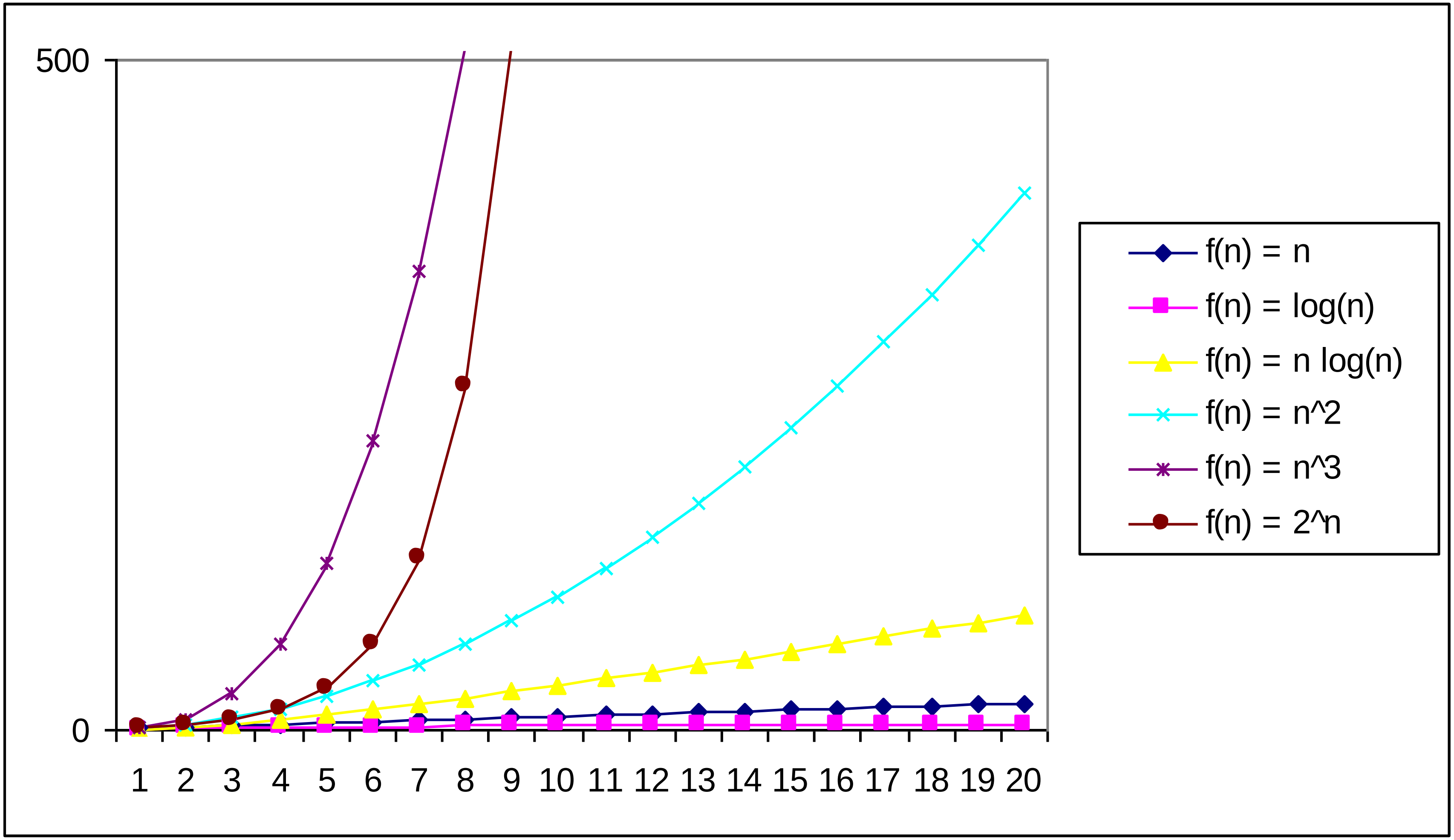
- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\le \quad n^k \sum a_i \frac{n^i}{n^k} \quad \le \quad n^k \sum a_i \quad \le \quad cn^k$$

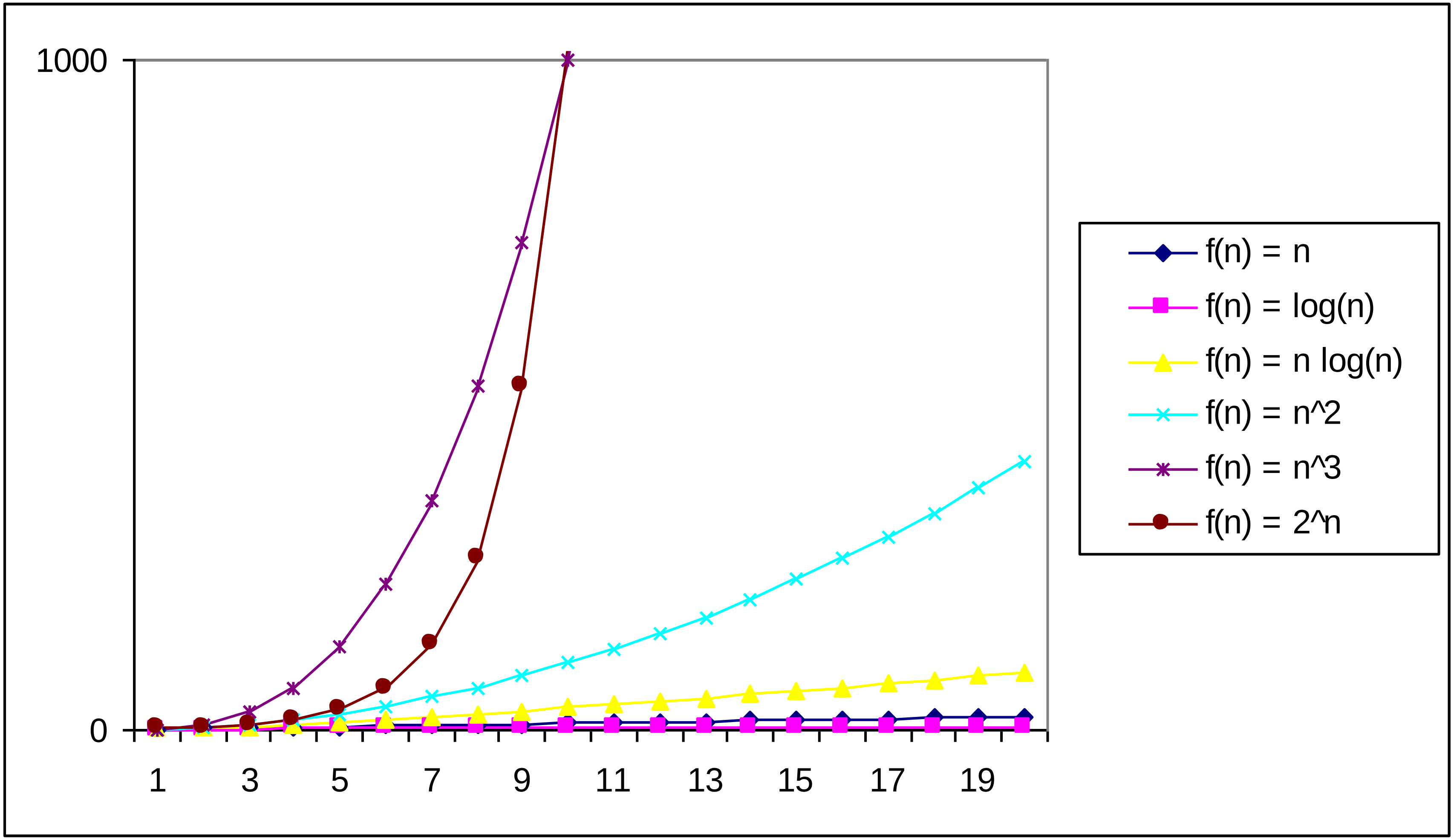
Practical Complexity



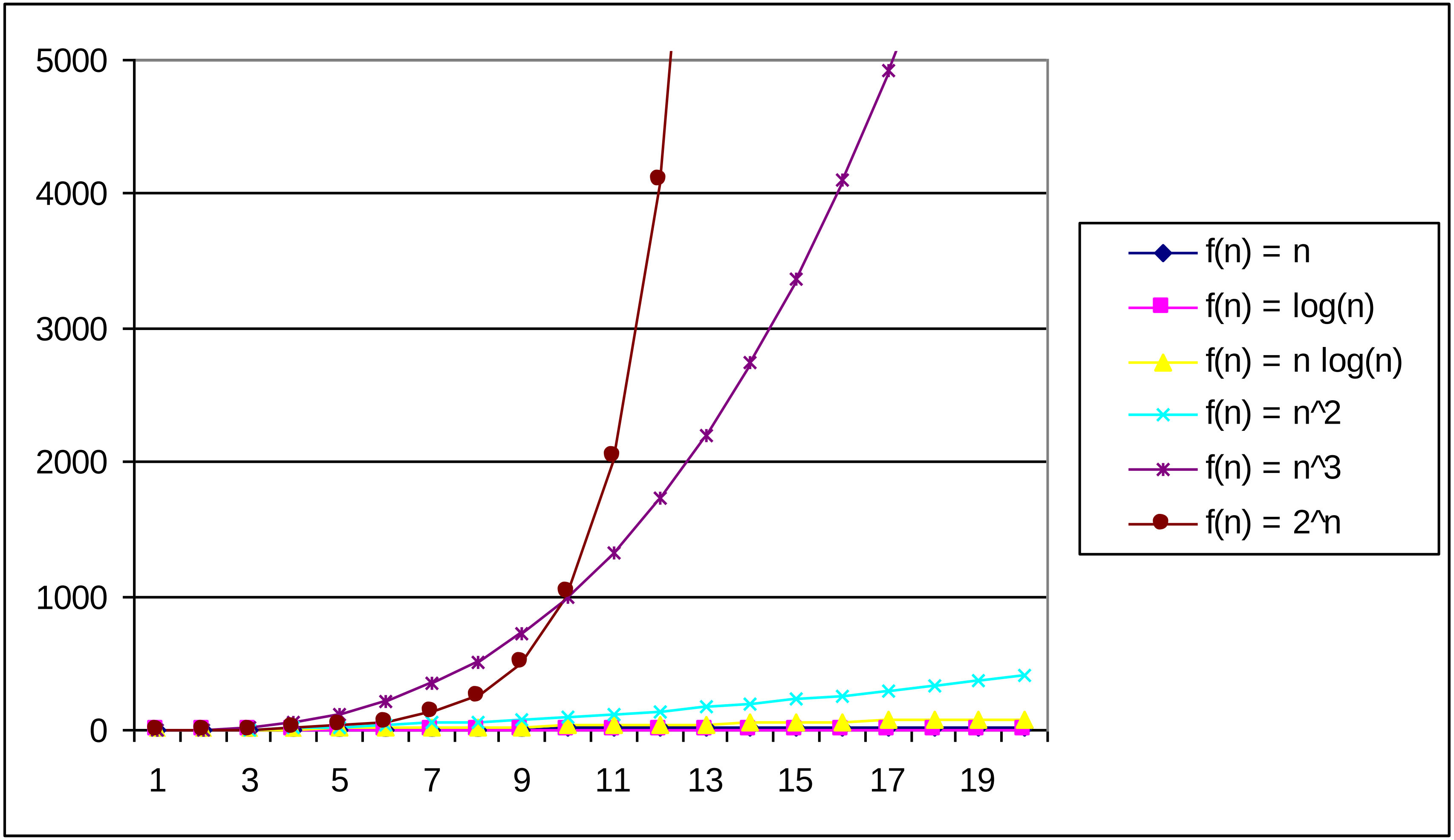
Practical Complexity



Practical Complexity



Practical Complexity



Asymptotic Performance

- In this course, we care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Pattern Matching

Outline and Reading

- Pattern matching algorithms
  - Brute-force algorithm
  - Boyer-Moore algorithm
  - Knuth-Morris-Pratt algorithm

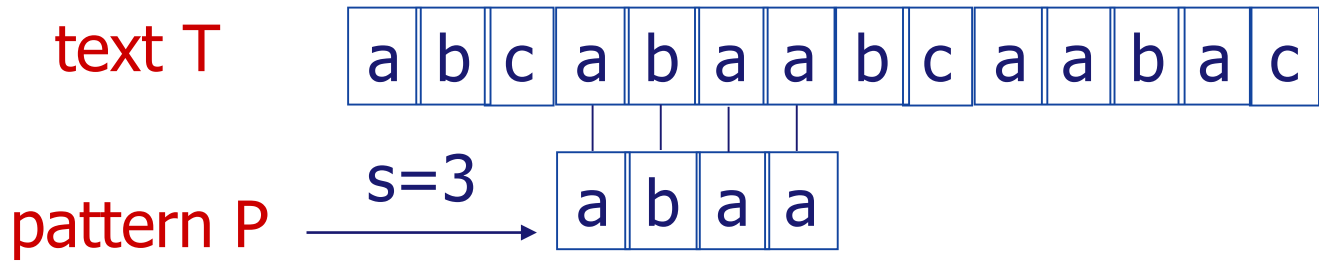
Strings

- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}

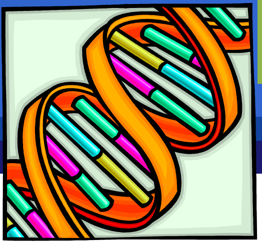
- Let P be a string of size m
  - A substring $P[i \,..\, j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type $P[0 \,..\, i]$
  - A suffix of P is a substring of the type $P[i \,..m - 1]$
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P
- Applications:
  - Text editors
  - Search engines
  - Biological research

String Matching

Given: Two strings T[1..n]] of length *n* and P[1..m] of length *m* over alphabet Σ
(The elements of *P* and *T* are characters drawn from a finite alphabet set *Σ*.)
Goal: Find all occurrences of P[1..m] "the pattern" in T[1..n] "the text".
Example: Σ = {a, b, c}



Brute-Force Algorithm



- The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text

Algorithm *BruteForceMatch*(T, P)
  Input text T of size n and pattern P of size m
  Output starting index of a substring of T equal to P or -1 if no such substring exists
  for $i \leftarrow 0$ **to** $n - m$
    { test shift i of the pattern }
    $j \leftarrow 0$
    while $j < m \wedge T[i + j] = P[j]$
      $j \leftarrow j + 1$
    if $j = m$
      return i {match at i}
    else
      break while loop {mismatch}
  return -1 {no match anywhere}